

Appendix A, example XMON source Code

XMON primarily consists of two parts. The first part handles the Debug exception of the Xtensa processor. This is implemented in the file `DebugExceptionVectorHandler-mon.S`. The second part is implemented in `xtensa-mon.c` and handles the higher level protocol with the debugger.

I. DebugExceptionVectorHandler-mon.S

```
// Exports
.global _DebugExceptionFromVector
.global _ar_registers
.global _sr_registers
.global _level_0_interrupt
.global _flush_i_cache
.global _xmon_out
.global _xmon_in
.global _xmon_flush
.global _xmon_init

// Imports
// _handle_exception

#include <machine/specreg.h>
#include "DebugExceptionVectorHandler.h"

#define AR_SAVE_SIZE (4*NUM_AREGS)
#define SR_SAVE_SIZE (4*256)

// Parameters
#define XMON_STACK_SIZE (2048+1024)
```

09680126-100300

093016-1030

In the comments below, we will use "ipwb" to refer to the interrupted process' window base, and "wb" to the current window base.

```
//      .section .bss
//      .section .text
//      .align 16
//_ar_registers:
//      .space AR_SAVE_SIZE
//_sr_registers:
//      .space SR_SAVE_SIZE
_xmon_stack:
      .space XMON_STACK_SIZE
_xmon_stack_bot:

      .text
      .begin literal
      .align 4
_xmon_stack_ptr:
      .word _xmon_stack_bot-4*16
ar_save_ptr:
      .word _ar_registers
sr_save_area_ptr:
      .word _sr_registers
//ar_save_area_ptr:
//      .word _registers+ARO_OFFSET

      .globl _handle_exception
handler:
      .word _handle_exception

      .align 4
.Laddress_of_savel_table_ptr:
      .word savel_table_ptr

      .align 4
savel_table_ptr:
      .word savel_28 /* ipwb=0 */
      .word savel_24 /* ipwb=1 */
      .word savel_20 /* ipwb=2 */
      .word savel_16 /* ipwb=3 */
      .word savel_12 /* ipwb=4 */
      .word savel_8 /* ipwb=5 */
      .word savel_4 /* ipwb=6 */
      .word savel_0 /* ipwb=7 */

      .align 4
.Laddress_of_save2_table_ptr:
      .word save2_table_ptr
save2_table_ptr:
      .word save2_0 /* ipwb=0 */
      .word save2_4 /* ipwb=1 */
      .word save2_8 /* ipwb=2 */
      .word save2_12 /* ipwb=3 */
      .word save2_16 /* ipwb=4 */
```


First, we save windows wb+1...n-1 where n == number of windows.
Second, we save windows 0,...,wb-1

```

/*
/* Disable WOE */
l32r    a3, .LDWOE
rsr     a2, PS
and     a2, a2, a3
wsr     a2, PS
rsync

addi    a0,a1,16      /* compute next save area */
rsr     a1,WINDOWBASE
slli    a1,a1,2
l32r    a2,.Laddress_of_save1_table_ptr
add     a2,a1,a2
l32i    a2,a2,0
jx      a2
/* The instruction jumps into the 1st part of the save sequence
   with the following notable register contents:
   a0 = ar_save_area_ptr + (ipwb+1)*16; i.e the save area for the
                                   next window.
   wb = ipwb
*/
save1_28: /* ipwb = 0 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save1_24: /* ipwb = 1 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save1_20: /* ipwb = 2 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save1_16: /* ipwb = 3 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save1_12: /* ipwb = 4 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save1_8: /* ipwb = 5 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save1_4: /* ipwb = 6 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
rotrw   1

```

```

save1_0: /* ipwb = 15 */
/* at this point, wb = 15; a0 = ar_save_area_ptr+n_aregs*4;
   i.e. a0 points to the end of the save area */
/* Now save 0...wb-1. i.e. the wrap around case */
l32r    a0,ar_save_ptr
l32r    a2,sr_save_area_ptr
l32i    a1,a2,(WINDOWBASE*4)      /* retrieve ipwb */
slli    a1,a1,2
l32r    a2,.Laddress_of_save2_table_ptr
add     a2,a1,a2
l32i    a2,a2,0
jx      a2
/* wb = 15; a0 = ar_save_area_ptr */
save2_28: /* ipwb = 7 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save2_24: /* ipwb = 6 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save2_20: /* ipwb = 5 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save2_16: /* ipwb = 4 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save2_12: /* ipwb = 3 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save2_8: /* ipwb = 2 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
addi    a4,a0,16
rotrw   1
save2_4: /* ipwb = 1 */
s32i    a4,a0,0
s32i    a5,a0,4
s32i    a6,a0,8
s32i    a7,a0,12
rotrw   1
save2_0: /* ipwb = 0 */
/* wb = (ipwb-1) mod (n_aregs/4) */
/* Now save special registers.
   We key it by testing the presence of register numbers.
   When present, the numbers indicate the user has configured
   the process to have the corresponding processor options.
   Note this doesn't quite work for TIE instructions yet.
*/
l32r    a0,sr_save_area_ptr

```

```

#define SAVE(r)          \
    rsr    a2, r;        \
    s32i    a2, a0, (r*4)

#ifdef ACCLO_OFFSET
    SAVE(ACCL0)
    SAVE(ACCHI)
    SAVE(MR_0)
    SAVE(MR_1)
    SAVE(MR_2)
    SAVE(MR_3)
#endif

#ifdef AV0_OFFSET
    SAVE(AVLO)
    SAVE(AVHI)
    SAVE(BV)
    SAVE(SAV)
#endif

#ifdef BR_OFFSET
    SAVE(BR)
#endif

    SAVE(CACHEATTR)

#ifdef CCOUNT_OFFSET
    SAVE(CCOUNT)
#endif

#ifdef CPENABLE_OFFSET
    SAVE(CPENABLE)
#endif

    SAVE(DEBUGCAUSE)
    SAVE(EPC_1)
    SAVE(EXCSAVE_1)
    SAVE(EXCCAUSE)
    SAVE(ICOUNT)
    SAVE(ICOUNTLEVEL)
    SAVE(INTENABLE)
    SAVE(INTREAD)
    SAVE(LBEG)
    SAVE(LCOUNT)
    SAVE(LEND)
    SAVE(SAR)

/* Disable Interrupts and Icounts */
movi.n a2, 0
wsr    a2, INTENABLE
wsr    a2, ICOUNTLEVEL
wsr    a2, ICOUNT
isync

/* Load new PS: Enable WOE and lower priority.
   We have already turned off interrupts and icount
   from above. */
l32r   a1, .Lps
wsr    a1, PS

movi.n a0, 0
movi.n a2, 1
wsr    a2, WINDOWSTART          /* window start = 1 */
wsr    a0, WINDOWBASE          /* window base = 0 */
rsync

/* Initialize our stack and call handler */
l32r   a1, _xmon_stack_ptr
l32r   a2, handler
callx4 a2

/* Raise interrupt level back up and disable WOE */
rsr    a2, PS
movi.n a3, 0
or     a2, a2, a3
l32r   a3, .LDWOE
and    a2, a2, a3

```

```

        wsr      a2, PS
        rsync

        /* restore sequence */
        l32r     a0, sr_save_area_ptr

#define RESTORE(r)          \
        l32i     a2, a0, (r*4); \
        wsr      a2, r

#ifdef ACCLO_OFFSET
        RESTORE(ACCLO)
        RESTORE(ACCHI)
        RESTORE(MR_0)
        RESTORE(MR_1)
        RESTORE(MR_2)
        RESTORE(MR_3)
#endif

#ifdef AV0_OFFSET
        RESTORE(AVLO)
        RESTORE(AVHI)
        RESTORE(BV)
        RESTORE(SAV)
#endif

#ifdef BR_OFFSET
        RESTORE(BR)
#endif

        RESTORE(CACHEATTR)

#ifdef CCOUNT_OFFSET
        RESTORE(CCOUNT)
#endif

#ifdef CPENABLE_OFFSET
        RESTORE(CPENABLE)
#endif

        RESTORE(EPC_1)
        RESTORE(EXCSAVE_1)
        RESTORE(EXCCAUSE)
        RESTORE(INTENABLE)
        RESTORE(INTREAD)
        RESTORE(LBEG)
        RESTORE(LCOUNT)
        RESTORE(LEND)
        RESTORE(SAR)

        /* Now restore all the ar's */
        l32i     a2,a0,(WINDOWBASE*4)
        wsr      a2,WINDOWBASE          /* set wb to ipwb */
        rsync
        l32r     a0,ar_save_ptr
        rsr      a2,WINDOWBASE
        slli     a2,a2,2                  /* multiply by 4 */
        l32r     a3,.Laddress_of_restorel_table_ptr
        add      a3,a2,a3
        l32i     a3,a3,0
        slli     a2,a2,2                  /* multiply by 4 */
        add      a8,a0,a2
        addi     a8,a8,16
        jx       a3
        /* wb = ipwb; a8 = ar_save_area_ptr + (ipwb+1)*16 */
restorel_28: /* ipwb = 0 */
        l32i     a4,a8,0
        l32i     a5,a8,4
        l32i     a6,a8,8
        l32i     a7,a8,12
        addi     a12,a8,16
        rotw     1
restorel_24: /* ipwb = 1 */
        l32i     a4,a8,0
        l32i     a5,a8,4
        l32i     a6,a8,8

```

```

132i      a7,a8,12
addi      a12,a8,16
rotrw     1
restorel_20: /* ipwb = 2 */
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
addi      a12,a8,16
rotrw     1
restorel_16: /* ipwb = 3 */
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
addi      a12,a8,16
rotrw     1
restorel_12: /* ipwb = 4 */
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
addi      a12,a8,16
rotrw     1
restorel_8: /* ipwb = 5 */
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
addi      a12,a8,16
rotrw     1
restorel_4: /* ipwb = 6 */
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
rotrw     1
restorel_0: /* ipwb = 15 */
/* wb = 15 */
/* restore window 0...wb-1 or none if wb == 0 */
132r      a4, sr_save_area_ptr
132i      a5,a4,(WINDOWBASE*4)
slli      a5,a5,2
132r      a6, .Laddress_of_restore2_table_ptr
add        a6,a5,a6
132i      a6,a6,0
132r      a4, ar_save_ptr
addi      a4,a4,-16
jx        a6
/* wb = 15; a4 = ar_save_area_ptr-16 */
restore2_28: /* ipwb = 7 */
addi      a8,a4,16
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
rotrw     1
restore2_24: /* ipwb = 6 */
addi      a8,a4,16
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
rotrw     1
restore2_20: /* ipwb = 5 */
addi      a8,a4,16
132i      a4,a8,0
132i      a5,a8,4
132i      a6,a8,8
132i      a7,a8,12
rotrw     1

```



```

restore2_16: /* ipwb = 4 */
    addi    a8,a4,16
    l32i    a4,a8,0
    l32i    a5,a8,4
    l32i    a6,a8,8
    l32i    a7,a8,12
    rotw    1
restore2_12: /* ipwb = 3 */
    addi    a8,a4,16
    l32i    a4,a8,0
    l32i    a5,a8,4
    l32i    a6,a8,8
    l32i    a7,a8,12
    rotw    1
restore2_8: /* ipwb = 2 */
    addi    a8,a4,16
    l32i    a4,a8,0
    l32i    a5,a8,4
    l32i    a6,a8,8
    l32i    a7,a8,12
    rotw    1
restore2_4: /* ipwb = 1 */
    addi    a8,a4,16
    l32i    a4,a8,0
    l32i    a5,a8,4
    l32i    a6,a8,8
    l32i    a7,a8,12
    rotw    1
restore2_0:
    l32i    a5,a4,20
    l32i    a6,a4,24
    l32i    a7,a4,28
    l32i    a4,a4,16
    rotw    1

    /* set wstart to what the user had */
    wsr     a0,EXCSAVE_0
    l32r    a0, sr_save_area_ptr

    l32i    a0, a0, (WINDOWSTART*4)
    wsr     a0,WINDOWSTART
    rsr     a0, WINDOWBASE
    wsr     a0, WINDOWBASE /* no-op but to avoid iss problem */

    l32r    a0, ar_save_ptr
    s32i    a1, a0, 0 /* save a1, we don't need loc anymore*/

    /* restore ICOUNT & ICOUNTLVL */
    l32r    a0, sr_save_area_ptr
    movi.n  a1, 0
    wsr     a1, ICOUNTLEVEL          // first lower icountlevel to 0
    isync
    l32i    a1, a0, (ICOUNT*4)
    wsr     a1, ICOUNT              // now write icount.
    isync
    l32i    a1, a0, (ICOUNTLEVEL*4)
    wsr     a1, ICOUNTLEVEL         // finally set icountlvl
    isync

    /* Enable WOE */
    //l32r    a1, .LEWOE
    //rsr     a0, PS
    //or      a0, a0, a1
    //wsr     a0, PS
    //rsync
    //l32r    a0, save_area_ptr

    // Put ar_save_area_ptr back into a0 so
    // that we can restore a1
    l32r    a0, ar_save_ptr
    l32i    a1, a0,0

```



```
# void _xmon_set_cpenable(unsigned value)
#
# a2 -- holds the value to set cpenable to
#
```

```
        .global _xmon_set_cpenable
        .align 4
_xmon_set_cpenable:
        entry    sp, 16
#ifdef CPENABLE_OFFSET
        wsr      a2, CPENABLE
        rsync
#endif
        retw
```

```
# void _xmon_set_user_register(unsigned user_register, unsigned value, unsigned *execute_here)
#
# a2 -- user_register
# a3 -- value
# a4 -- pointer to memory to execute from
```

```
#
#
        .align 4

.wur0_instruction:
        .word    0x00f30000
.wur0_insn_ptr:
        .word    .wur0_instruction

.wur0_placeholder_ptr:
        .word    .wur0_instruction_placeholder
```

```
        .align 4
.global _xmon_set_user_register
_xmon_set_user_register:
        entry    sp, 48
```

```
# a6 -- temporary for moving memory
# a5 -- pointer to wur0_placeholder
# a4 -- points to the RAM location we will
#       execute from, move the base instruction
#       (including the retw) to that point.
```

```
        l32r     a5, .wur0_placeholder_ptr
        l32i     a6, a5, 0
        s32i     a6, a4, 0
        l32i     a6, a5, 4
        s32i     a6, a4, 4
```

```
# a5 -- available again, now used to load the
#       base wur instruction which we will now
#       modify for the correct ar and user register
#       number
# a6 -- holds the modified instruction
```

```
        l32r     a5, .wur0_insn_ptr
        l32i     a6, a5, 0
```

```
# a2 -- holds the user register we are going to write
# a4 -- holds the location in memory that we are going
#       to execute from
# a6 -- holds the instruction we are going to execute
```

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000



09680126-100300

```

        slli    a2, a2, 8
        or      a6, a6, a2

# a2 -- Can be used as a temporary now, to
#       OR in the 3, which is the register that
#       holds the value we are going to write to
#       the user register

        movi    a2, 3
        slli    a2, a2, 4
        or      a6, a6, a2

# a2 -- Temporary for merging instructions
# a4 -- pointer to the location we are going to execute
#       from
# a5 -- Holds the value we load from our execution point
# a6 -- The instruction that we are going to execute

        l32i    a5, a4, 0

# Need to merge our 24-bit instruction with 8 bits
# from our execute point
# Want to use the lower 24 bits from a6,
# and the upper 8-bits from a5

        movi    a2, 0xff
        slli    a2, a2, 24
        and     a5, a5, a2
        or      a6, a6, a5
        s32i    a6, a4, 0

# Flush the cache
        mov     a10, a4
        call8   _flush_i_cache

        jx      a4

# Want the upper 24-bits from a6, and the
# lower 8-bits from a4

.align 4
.wur0_instruction_placeholder:
        or      a0, a0, a0
        retw

#
#
# Data for _xmon_get_user_register
#
        .align 4
.rur0_insn:
        .word   0x00e30000

.rur0_insn_ptr:
        .word   .rur0_insn

.rur_placeholder_ptr:
        .word   .rur_instruction_placeholder

# unsigned int _xmon_get_user_register(unsigned user_register, unsigned *execute_here)
#

```

```

# a2(input) -- user_register
# a2(output) -- contains the value
# a3(input) -- address for executing instructions

.align 4
.global _xmon_get_user_register
_xmon_get_user_register:
    entry sp, 48

# a5 -- temporary for moving memory
# a4 -- Points to our rur instruction including ret
# that we are going to copy to the execution point
# a3 -- Points to the execution point

    l32r a4, .rur_placeholder_ptr
    l32i a5, a4, 0
    s32i a5, a3, 0
    l32i a5, a4, 4
    s32i a5, a3, 4

# a4 -- Temp that Points to the rur0 instruction
# a6 -- will hold the rur instruction throughout

    l32r a4, .rur0_insn_ptr
    l32i a6, a4, 0

# Shift the user register number to the correct
# offset and OR it into our instruction
# a2 -- Holds the user register being read
# a6 -- instruction being massaged

    slli a2, a2, 4
    or a6, a6, a2

# Now need to set the r-field of the instruction
# to be 2, which is the return value of this function
# a5 -- Temp that holds the constant being ord in
# a6 -- The instruction being massaged

    movi a5, 2
    slli a5, a5, 12
    or a6, a6, a5

# Now load in the word from where we are going to execute
# the rur, merge our rur instruction, and store that word
# back to memory.
# a2 -- Temp for masking
# a3 -- Points to the correct memory location
# a5 -- Holds the WORD we are manipulating

    l32i a5, a3, 0

# In Little Endian we save the MSB and put our
# instruction in the lower 3 bytes

    movi a2, 0xff
    slli a2, a2, 24
    and a5, a5, a2
    or a6, a6, a5
    s32i a6, a3, 0

# Clear the cache line
# a3 -- Points to the location being cleared

    mov a6, a3
    call4 _flush_i_cache

    movi a2, 0
    jx a3

```

00000000 00000000 00000000 00000000


```

*
*   c           Resume at current address           SNN   ( signal NN)
*   CAA..AA     Continue at address AA..AA          SNN
*
*   s           Step one instruction               SNN
*   sAA..AA     Step one instruction from AA..AA    SNN
*
*   k           kill
*
*   ?           What was the last signal ?          SNN   (signal NN)
*
*   bBB..BB     Set baud rate to BB..BB            OK or BNN, then sets
*                                           baud rate
*
* All commands and responses are sent with a packet which includes a
* checksum. A packet consists of
*
* $<packet info>#<checksum>.
*
* where
* <packet info> :: <characters representing the command or response>
* <checksum>    :: < two hex digits computed as modulo 256 sum of <packetinfo>>
*
* When a packet is received, it is first acknowledged with either '+' or '-'.
* '+' indicates a successful transfer. '-' indicates a failed transfer.
*
* Example:
*
* Host:                      Reply:
* $m0,10#2a                  +$00010203040506070809101112131415#42
*
* *****/

```

```

#include <stdio.h>
#include <signal.h>
#include <machine/specreg.h>
#include <machine/xtl000.h>
#include "DebugExceptionVectorHandler.h"
#include "uart.h"
#include "xtensa-libdb.h"

#define WS_MASK (~(0)<<(NUM_AREGS/4))
#ifdef IS_LITTLE_ENDIAN

#define IS_BREAKN(p) ((p)[0]==0x2d && ((p)[1]&0xf0)==0xf0)
#define IS_BREAK(p) ((p)[2]==0x00 && ((p)[0]&0xf0)==0x00 && ((p)[1]&0xf0)==0x40)
#define BREAKNO(p) (IS_BREAKN(p) ? ((p)[1]&0xf0) : -1 )
#define BREAK_S(p) ((p)[1]&0xf0)
#define BREAK_T(p) (((p)[0]&0xf0)>>4)
#else

#define IS_BREAKN(p) ((p)[0]==0xd2 && ((p)[1]&0xf0)==0xf0)
#define IS_BREAK(p) ((p)[2]==0x00 && ((p)[Q]&0xf0)==0x00 && ((p)[1]&0xf0)==0x04)
#define BREAKNO(p) (IS_BREAKN(p) ? (((p)[1]&0xf0)>>4) : -1 )
#define BREAK_S(p) (((p)[1]&0xf0)>>4)
#define BREAK_T(p) ((p)[0]&0xf0)

#endif

#define SR_REG(n) (_sr_registers[ (n) ])

/* Macros to extract fields of PS */
#define GET_PSINTLVL(ps) ((ps)&0xf)
#define GET_PSUSRMODE(ps) (((ps)>>5)&0x1)
#define GET_PSOWB(ps) (((ps)>>8)&0xf)
#define GET_PSCALLINC(ps) (((ps)>>16)&0x3)
#define GET_PSWOE(ps) (((ps)>>18)&0x1)

/* Imported functions */
extern void _flush_i_cache( char *);
extern int _xmon_out(char c);

```



```

/* Parameters:
   We'll need to create a configuration process that generates
   many of these defines.
*/
/*****
/* BUFMAX defines the maximum number of characters in inbound/outbound buffers*/
/* at least NUMREGBYTES*2 are needed for register packets */
#define BUFMAX 2048

#define AR_SAVE_SIZE (4*NUM_AREGS)
#define SR_SAVE_SIZE (4*256)

#define PC EPC_0
#define DBG_EPS EPS_0

long _ar_registers[AR_SAVE_SIZE/(sizeof (long))];
long _sr_registers[SR_SAVE_SIZE/(sizeof (long))];

/* !0 means we are running on the board, defined by linker */
extern void *IN_SIMULATOR;
int _in_simulator = (int)&IN_SIMULATOR;
int _initialized = 0; /* !0 means we've been initialized */

static const char hexchars[]="0123456789abcdef";

/* string functions */
int _strlen( char *cp )
{
    int i;
    if( cp == 0 )
        return 0;
    i = 0;
    while ( *cp )
    {
        i++;
        cp++;
    }
    return i;
}

char *_strcpy( char *d, char *s )
{
    char *cp = d;
    if( d && s )
    {
        while( *s )
        {
            *cp = *s;
            cp++;
            s++;
        }
        *cp = 0;
    }
    return d;
}

void _memset(unsigned char *ptr, unsigned char value, int num)
{
    while (num > 0)
    {
        *ptr = value;
        ++ptr;
        --num;
    }
}

```

09680126 = 1003000


```

static void
putpacket(buffer)
    unsigned char *buffer;
{
    unsigned char checksum;
    unsigned char ack;
    int count;
    unsigned char ch;

    /* $<packet info>#<checksum>. */
    do
    {
        putDebugChar('$');
        checksum = 0;
        count = 0;

        while (ch = buffer[count])
        {
            putDebugChar(ch);
            checksum += ch;
            count += 1;
        }

        putDebugChar('#');
        putDebugChar(hexchars[checksum >> 4]);
        putDebugChar(hexchars[checksum & 0xf]);

        flushDebug();
        ack = getDebugChar();
        ack = ack & 0x7f;

//      led_display_ok();
/*
        if (ack != '+')
        {
            //      char buf[8];
            //      _memset(buf, 0, sizeof(buf));
            //      putDebugString("--");
            //      mem2hex(&ack, buf, 1, 0);
            putDebugString(buf);
            putDebugString("--");
        }
        else
            putDebugChar('Y');
    */
    }
    while (ack != '+');
}

static char remcomInBuffer[BUFMAX];
static char remcomOutBuffer[BUFMAX];

static unsigned char g_execute_here[1024];

static void bad_protocol()
{
    _strcpy( remcomOutBuffer, "Error: garbled command" );
}

static void aok()
{
    _strcpy( remcomOutBuffer, "OK" );
}

/* Decode a hex string and write it into memory. */
static char *
write_mem(buf, mem, count, flush, verify)
    register unsigned char *buf;
    register unsigned char *mem;

```



```

#endif
    _flush_i_cache(b->address);
    _flush_i_cache(b->address+1);
    _flush_i_cache(b->address+2);
#endif
    b++;
}

static void clear_special_breaks()
{
    struct special_breakpoint *b;
    b = special_break;
    while( b->address )
    {
#ifdef ISAUSEDENSITYINSTRUCTION
        b->address[0] = b->saved_inst[0];
        b->address[1] = b->saved_inst[1];
        _flush_i_cache(b->address);
        _flush_i_cache(b->address+1);
#else
        b->address[0] = b->saved_inst[0];
        b->address[1] = b->saved_inst[1];
        b->address[2] = b->saved_inst[2];
        _flush_i_cache(b->address);
        _flush_i_cache(b->address+1);
        _flush_i_cache(b->address+2);
#endif
        b++;
    }
}

// !!@

static void do_special_breaks(int *state, int *sigval)
{
    char *pc;
    struct special_breakpoint *b;

    b = special_break;
    pc = (char *)SR_REG(PC);

    while( b->address )
    {
        if( b->address == pc && b->f )
        {
            b->f(state, sigval);
            return;
        }
        b++;
    }

    *state = XMON_CONTROL;
    *sigval = SIGTRAP;
}

#if 0
void set_ps( int eps )
{
    REG(PSINTLVL) = GET_PSINTLVL(eps);
    REG(PSUSRMODE) = GET_PSUSRMODE(eps);
    REG(PSOWB) = GET_PSOWB(eps);
    REG(PSCALLINC) = GET_PSCALLINC(eps);
    REG(PSWOE) = GET_PSWOE(eps);
}
#endif

```

```

#if 0
void flash_value(unsigned int value)
{
    int i = 0;

    for(i = 28; i >= 0; i=i-4)
    {
        int number = (value >> i) & 0xf;

        led_blank();
        led_pause(100000);
        led_display_digit(number);
        led_pause(100000);
    }
}
#endif

void setup_ps()
{
    /* the code to set up the PS works quite differently depending
       on whether or not the uart is on interrupt level one. */
    #if UART_INTERRUPT_LEVEL == 1
    {
        unsigned int real_ps = 0;
        real_ps = SR_REG(EPS_0);

        // We can figure out if PS.UM was set by looking
        // at which vector we came from
        // UART_VECTOR is actually the UserExceptionVector

        if ( SR_REG(UART_EPC) == UART_VECTOR )
        {
            // Since we are coming from UserExceptionVector
            // turn on the PS.UM mode bit.
            real_ps = real_ps | (1 << 4);

            // Assume that WOE is always enabled for user code.
            real_ps = real_ps | (1 << 17);
        }
        else
        {
            // We are coming from the KernelExceptionVector
            // So we leave PS.UM disabled, and we take
            // WOE from the current PS.
            real_ps = real_ps | ( SR_REG(EPS_0) & (1 << 17) );
        }

        // Set interrupt level to 0
        real_ps = real_ps & ~0xf;
        SR_REG(EPS_0) = real_ps;
    }
    #elif UART_INTERRUPT_LEVEL != -1
        SR_REG(EPS_0) = SR_REG(UART_EPS);
    #endif
}

/* If we see a break on the UART then we simulate a SIGINT */
static void sigint_handler( int *state, int *sigval )
{
    int interrupts = SR_REG(INTERRUPT);
    int c;

    // led_display_digit(7);
    // led_pause(100000);

    // flash_value(interrupts);

```



```

        /* We're running on the board, so flash the status
        LEDs a few times */
        _uart_init((_uart_dev_t *)XT1000_DUART_1_ADDR, B38400);
        _uart_enable_rcvr_int((_uart_dev_t *)XT1000_DUART_1_ADDR);
        led_display_ok();
        led_display_digit(2);
//
    }
    else
    {
        _xmon_init();
    }
    putDebugString("XMON R2.5 ");
    putDebugString(_in_simulator ? " running on iss\n"
        : " running on eval board\r\n");

    _initialized = 1;

    init_special_breaks();
    state = XMON_CONTROL;
    continue;

case XMON_CONTROL:    /* let host control us */
    _tell_gdb( signal );
    set_special_breaks();
    state = XMON_RUNNING;
    return;            /* return to user program */

case XMON_RUNNING:
    if(IS_BREAK(pc)) {
        unsigned s = BREAK_S(pc);
        unsigned t = BREAK_T(pc);
        if( s==1 && (t <= 1)) {
            switch(SR_REG(EXCCAUSE)) {
                case EXCCAUSE_ILLEGAL:
                    signal = SIGILL;
                    break;
                case EXCCAUSE_SYSCALL:
                    signal = SIGTRAP;
                    break;
                case EXCCAUSE_IFETCHERROR:
                case EXCCAUSE_LOADSTOREERROR:
                    signal = SIGSEGV;
                    break;
                case EXCCAUSE_LEVEL1INTERRUPT:
                    signal = SIGINT;
                    break;
            }
            clear_special_breaks();
            state = XMON_CONTROL;

            /* pretend as though we caught it
            at the point of occurrence */
            // !!@
            SR_REG(PC) = SR_REG(EPC_1);
            setup_ps();
            continue;
        }
    }
    n = BREAKNO(pc);
    switch(n)
    {
    default:
        clear_special_breaks();
        state = XMON_CONTROL;
        break;

    case 1:            /* special breakpoint */
        clear_special_breaks();

        /* keep in mind that the state can be changed by the
        do_special_breaks handler. */

```

```

        do_special_breaks(&state, &sigval);
        break;
    }
    continue;

case XMON_RESUMING:
    /* we have taken an interrupt that was not the serial
       interrupt and have now executed the original instruction
       at the interrupt vector. Now restore the break instruction
       so that interrupts continue to work and resume. */

    set_special_breaks();
    state = XMON_RUNNING;
    return;
}
}

/*
 * This function does all command processing for interfacing to gdb.
 */
unsigned char dummy[4];
unsigned int  user_register_value;

unsigned int execution_space[2];

static unsigned char *
get_reg_ptr(const unsigned int libdb_target_number)
{
    unsigned char *reg_ptr = NULL;
    unsigned      old_cpe  = 0;
    int           offset   = 0;

    switch( GET_TARGET_REG_TYPE(libdb_target_number) )
    {
        case REGTYPE_AR:
            offset = GET_TARGET_REG_INDEX(libdb_target_number);
            reg_ptr = (unsigned char *)&_ar_registers[offset];
            break;

        case REGTYPE_SPECIAL_REG:
            offset = GET_TARGET_REG_INDEX(libdb_target_number);
            reg_ptr = (unsigned char *)&_sr_registers[offset];
            break;

        case REGTYPE_USER_REG:
            old_cpe = _xmon_get_cpenable();
            _xmon_set_cpenable((unsigned)-1);
            user_register_value = _xmon_get_user_register( GET_TARGET_REG_INDEX(libdb_target_number),
                                                         &execution_space[0] );
            _xmon_set_cpenable(old_cpe);
            reg_ptr = (unsigned char *)&user_register_value;
            break;

        default:
            reg_ptr = NULL;
            break;
    }

    return reg_ptr;
}

static unsigned int
set_reg_value(const unsigned int libdb_target_number, const unsigned int value)
{
    int success = 0;

    if ( GET_TARGET_REG_TYPE(libdb_target_number) == REGTYPE_USER_REG )
    {

```


090301-100000

0960166-100900


```

        // !!@
        //if(mem2hex( (char *)&_registers[addr], remcomOutBuffer, 4, 0 ))
        //    break;
        reg_ptr = get_reg_ptr( addr);
        if (reg_ptr != NULL)
        {
            if(mem2hex( reg_ptr, remcomOutBuffer, 4, 0 ))
                break;
        }
        _strcpy( remcomOutBuffer, "Error: read failure" );
    }
    else
        bad_protocol();
    break;

case 'P':
    ptr = &remcomInBuffer[1];
    if( hexToInt(&ptr, &addr) && *ptr++ == '='
        && write_mem( ptr, (char *)&value, 4, 0, 0 ))
    {
        unsigned char *reg_ptr = NULL;
        if (set_reg_value(addr, value))
        {
            aok();
        }
        else
        {
            _strcpy( remcomOutBuffer, "Error: write failure" );
        }
    }
    // !!@
    //    _registers[addr] = value;
}
else
    bad_protocol();
break;

case 'm':          /* MAA..AA,LLLL Read LLLL bytes at address AA..AA */
/* Try to read %x,%x. */

    ptr = &remcomInBuffer[1];

    if (hexToInt(&ptr, &addr)
        && *ptr++ == ','
        && hexToInt(&ptr, &length))
    {
        if (mem2hex((char *)&addr, remcomOutBuffer, length, 1))
            break;
        _strcpy( remcomOutBuffer, "Error: read failure" );
    }
    else
        bad_protocol();
    break;

case 'M': /* MAA..AA,LLLL: Write LLLL bytes at address AA.AA return OK */
/* Try to read '%x,%x:'. */

    ptr = &remcomInBuffer[1];

    if (hexToInt(&ptr, &addr)
        && *ptr++ == ','
        && hexToInt(&ptr, &length)
        && *ptr++ == ':')
    {
        if (write_mem(ptr, (char *)&addr, length, 1, 1))
            aok();
        else
            _strcpy(remcomOutBuffer, "Error: write failure");
    }
    else
        bad_protocol();

```

```

        break;

case 'c':    /* CAA..AA  Continue at address AA..AA(optional) */
    /* try to read optional parameter, pc unchanged if no parm */

    ptr = &remcomInBuffer[1];
    if (hexToInt(&ptr, &addr))
    {
        SR_REG(PC) = addr;
    }
    //      else
    //      bad_protocol();
    return;

case 's':
    /* use icount mechanism to step a single instruction */
    set_icount_for_single_step(intlevel);
    return;

/* kill the program */
case 'k':
    /* do nothing */
    break;

/* Xtensa specific commands */
case 'X':
    switch( remcomInBuffer[1] )
    {
        case 'q':
            switch (remcomInBuffer[2])
            {
                case 'n':
                    _strcpy( remcomOutBuffer, "XMON2.5" );
                    break;

                case 'p':
                    _strcpy( remcomOutBuffer, "n");
                    break;

                case 'P':
                    _strcpy( remcomOutBuffer, "n");
                    break;

                default:
                    break;
            }
            break;

        case 'e':
            if ( remcomInBuffer[2] == 'x' && remcomInBuffer[3] == 'e' ).
            {
                ExecuteSomeInstruction( &remcomInBuffer[4] );
                remcomOutBuffer[0] = '\0';
            }
            break;

        case 'B':
            /* Set a breakpoint using the ibreak registers */
            #if NIBREAK==0
                _strcpy( remcomOutBuffer, "Error: configuration has no IBREAK registers");
            #else
                ptr = &remcomInBuffer[4];
                if( !hexToInt(&ptr, &addr ) ) {
                    bad_protocol();
                    break;
                }

                switch( remcomInBuffer[2] ) {
                    case 's':
                        /* set */
                        for( i = 0, bp = hw_break; i < NIBREAK; i++, bp++ )

```



```

        if( bp->free ) {
            bp->free = 0;
            bp->addr = (char *)addr;
            // !!@
            //_registers[bp->reg_number] = addr;
            SR_REG(IBREAKENABLE) |= (1<<i);
            aok();
            break;
        }
        if( i >= NIBREAK )
            _strcpy( remcomOutBuffer, "Error: out of ibreak registers");
        break;
    case 'r':
        /* remove */
        for( i = 0, bp = hw_break; i < NIBREAK; i++, bp++ )
            if( !bp->free && bp->addr == (char *)addr ) {
                bp->free = 1;
                bp->addr = 0;
                // !!@
                //_registers[bp->reg_number] = 0;
                SR_REG(IBREAKENABLE) &= ~(1<<i);
                aok();
                break;
            }
        if( i >= NIBREAK )
            _strcpy( remcomOutBuffer, "Error: breakpoint not found");
        break;
    default:
        break;
}
#endif
    }
    break;
#endif

    /* Test feature */
    case 't':
        asm (" std %f30,{%sp}");
        break;
    /* Reset */
    case 'r':
        asm ("call 0
             nop ");
        break;
#endif

    /* Disabled until we can unscrew this properly */
    case 'b':
        /* BBB... Set baud rate to BB... */
        {
            int baudrate;
            extern void set_timer_3();

            ptr = &remcomInBuffer[1];
            if (!hexToInt(&ptr, &baudrate))
            {
                _strcpy(remcomOutBuffer,"B01");
                break;
            }
        }

        /* Convert baud rate to uart clock divider */
        switch (baudrate)
        {
            case 38400:
                baudrate = 16;
                break;
            case 19200:
                baudrate = 33;
                break;
            case 9600:
                baudrate = 65;
                break;
            default:
                _strcpy(remcomOutBuffer,"B02");
        }

```

```

        goto x1;
    }

    putpacket("OK"); /* Ack before changing speed */
    set_timer_3(baudrate); /* Set it */
}

x1: break;

#endif

/* switch */

/* reply to the request */
putpacket(remcomOutBuffer);
}
}

```

SECRET

```

/*.....
Xtensa hardware-dependent utilities
.....*/

/* retrieved register at a particular window base */
static long reg_at_wb( unsigned int reg, unsigned int wb )
{
    unsigned int relocated_reg;
    if( (NUM_AREGS/4) <= wb )
        mon_error( "invalid window base in reg_at_wb\n" );
    if( NUM_VISIBLE_AREGS <= reg )
        mon_error( "invalid register in reg_at_wb\n" );
    relocated_reg = (reg + (wb<<2)) & AREGS_MASK;
    // relocated_reg += ARO_OFFSET/4;

    return _ar_registers[relocated_reg];
}

/* retrieved register in window of interrupt process */
static int reg_at_ipwb( unsigned int reg )
{
    return reg_at_wb( reg, SR_REG(WINDOWBASE) );
}

static int save_to_stack()
{
    int i;
    int ws = SR_REG(WINDOWSTART);
    int wb = SR_REG(WINDOWBASE);
    int callee_win, win;
    long *sp, *caller_sp;

    /* rotate so that first bit of ws corresponds to
       wb+1 */
    ws = (ws >> (wb+1)) | (ws << (NUM_AREGS/4-(wb+1)));
    ws &= WS_MASK;

    /* find first window after ipwb */
    if( ws == 0 )
        mon_error( "window start zero in save_to_stack\n" );
    for( i = 0; (ws&1)==0; i++ )
        ws >>= 1;
    ws >>= 1;
    i++;
    while( ws != 0 )
    {
        win = (wb+i) & WB_MASK;
        if( ws & 1 )
        {
            callee_win = (win+1) & WB_MASK;
            sp = (long *)reg_at_wb( 1, callee_win ) - 4;
            sp[0] = reg_at_wb( 0, win );
            sp[1] = reg_at_wb( 1, win );
            sp[2] = reg_at_wb( 2, win );
            sp[3] = reg_at_wb( 3, win );
            i = i+1;
            ws >>= 1;
            continue;
        }
        if( ws & 2 )
        {
            callee_win = (win+2) & WB_MASK;
            sp = (long *)reg_at_wb( 1, callee_win ) - 4;
            sp[0] = reg_at_wb( 0, win );
            sp[1] = (long) caller_sp = (long *)reg_at_wb( 1, win );
            sp[2] = reg_at_wb( 2, win );
            sp[3] = reg_at_wb( 3, win );
        }
    }
}

```



```
static int fetchUserRegister()
{
    __asm__("nop");
    return;
}
```

```
static void setUserRegister()
{
    __asm__("nop");
    return;
}
```

00E00F" 9270B960